Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1804 – 1813

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Implementation of Intel restricted transactional memory ISA extension in Simics

Grigory Rechistov[a,b], Arnold Plotkin[a,b]

[a] *Intel Supercomputer Applications Laboratory for Advanced Research, Moscow Institute of Physics and Technology, Russia*
[b] *Intel Corporation*

Abstract

Hardware transactional memory is finally becoming available in products from major vendors. Recently Intel announced that a set of transactional synchronization extensions (TSX) will be available in its next processor microarchitecture, codenamed Haswell. The benefits of software simulation of this technology will remain significant even after processors that support new instructions are available on the market. The reason for this is that a simulation often provides more flexibility during debugging and architecture exploration. In this paper we describe an implementation of Intel® restricted transactional memory (RTM) instructions, which are a part of the Intel® TSX, in the full system functional simulator Wind River® Simics. Our goal was to enable correct execution of these new instructions during all stages of operating system boot and user-level application execution and at the same time to keep the high simulation speed that Simics is able to demonstrate. This model is used to enable pre-silicon software development.

Keywords: simulation, Simics, hardware transactional memory, RTM, TSX

1. Introduction

Hardware transactional memory [1] (HTM) is finally becoming available in products from major vendors. IBM was the first to offer HTM in BlueGene/Q [2] and zEnterprise EC12 server [3]. Recently Intel announced that a set of transactional synchronization extensions (TSX) will be available in its next microprocessor architecture codenamed Haswell [4]. The benefits of software simulation of this technology will remain significant even after processors that support new instructions are available on the market. The reason for this is that a simulation often provides more flexibility during debugging and architecture exploration.

Intel® TSX poses certain challenges for a functional simulator designers who usually tend to omit modeling a cache hierarchy altogether for the sake of speed. A naïve implementation can break many original optimizations used in simulators and thus reduce their performance. To avoid such negative effects RTM features should be switched off completely when none of simulated threads are inside transactions. This would guarantee that new functionality does not affect the simulator operation when it is not needed.

The specification of Intel® TSX is public [5, chapter 8]. However, it lacks essential details of its hardware architecture and descriptions of exact conditions when certain events should or should not occur; that is, there are many "may" clauses in it. This uncertainty requires a reconfigurable solution that can be configured to match real systems when their exact specifications become available and also to explore available architectural options.

Email addresses: grigory.rechistov@phystech.edu (Grigory Rechistov), arnold.l.plotkin@intel.com (Arnold Plotkin)

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.349

The documentation only gives a description of an external interface and accepted behavior and does not describe details that are important for a model implementer. Many essentially different implementations conforming to it can exist. Considering the "volatile" nature of RTM (lack of any guarantee that a transactional path will progress), the simplest and the least useful one would just cancel any transaction right at its beginning. Therefore, to provide something more reasonable and practically useful we made decisions based on practical usage cases and our previous experience with RTM implementations. A significant part of internal CPU configuration related to transactional state was made visible to the end user so that she can adjust it to her needs. We wanted to create a model that adhered to the documentation, was useful in the sense that transactions have chances to commit and also required minimal modifications to the production-quality code of existing processor models.

The rest of the paper is organized as follows. Section 2 gives an overview of Wind River® Simics features relevant to the object of study. Section 3 presents a quick introduction of new instructions. Section 4 outlines the implementation. Section 5 uncovers issues that were discovered in the course of the research. Section 6 describes our evaluation with benchmark applications. Section 7 gives overview of related work. Finally in section 8 we make conclusions and outline future work.

2. Overview of Wind River® Simics

Wind River® Simics [6] is a fast functional simulator that is used to model full systems that consist of multiple processors, memory, peripheral devices like SATA disks and network cards, graphics cards etc. It is accurate enough to be able to boot unmodified operating systems such as different flavours of GNU/Linux, Microsoft Windows, VxWorks etc. It should be mentioned that Simics is not cycle accurate, i.e., it does not model internals of the CPU such as pipelines, out of order features, branch predictors and the like; by default it does not simulate caches.

Considering instruction set simulation of IA-32 architecture, it provides models for most of Intel ISA extensions, including vector commands of SSE, AVX and AVX2, virtualization technology Intel® VTx, trusted execution technology Intel® TXT etc. Simics CPU models are fast and extensible at the same time, because for every model there are up to three simulation engines that are dynamically switched during a simulation transparantely to a user:

1. Interpreter is the most flexible mode but also the slowest one.
2. Just in time compilation of target to host instructions, a technique also known as binary translation. This mode is generally faster than interpretation but is also more complex for adopting new instructions.
3. VMP mode leverages hardware virtualization support of Intel® VTx [7] technology to run target IA-32 code directly on host, falling back to JIT or interpreter at virtual machne exit events i.e., when it cannot directly execute new instructions not present on host or when an interrupt occurs. It is the fastest engine, though it is also the most complex one to extend and to debug because of a host kernel module required for VMP to operate. It is enabled only for simulation intervals that are longer than a certain threshold value because operations of entering and exiting the virtual monitor mode are time consuming and such frequent switches can nullify any performance benefits.

3. Overview of RTM ISA

3.1. New instructions

Intel® RTM extension to the IA-32 architecture includes four new instructions to start, stop, cancel and inquire status of transactions [5]. Below is a short description of them.

- `XBEGIN label` designates entering a transaction. The processor thread saves its registers in a checkpoint and goes to speculative mode. Further memory stores issued by this thread do not reach physical memory but are temporarily kept until either `XEND` is encountered or a disruptive event or instruction cancels the whole transaction. In the latter case the register state is restored from the checkpoint, instruction pointer (RIP register) is set to `label` and no memory write is done.
- `XEND` designates a successfull end of a transaction. If speculative execution reaches this instruction then all delayed memory stores are atomically committed. Saved register state checkpoint is discarded.

- `XABORT condition` cancels a current transaction. Thread's registers are restored from the checkpoint, except for the `EAX` that stores `condition` value, and instruction pointer that is set to `label` from the preceding `XBEGIN`.
- `XTEST` – modifies the CF flag to indicate whether the thread is inside a transaction or not.

While the documentation allows for nested transactions it demands that they are effectively "flattened" to the outermost one, that is, all inner `XBEGIN` and `XEND` instructions do not create or commit any additional speculative state.

3.2. Changes in instructions semantics

The RTM specification also dictates that several instructions are not allowed to be executed while a thread is inside a transaction – they would cancel it. They are `CPUID`, `PAUSE` and `XABORT`, the last one allows to specify a reason.

It is also clear that any instruction that may operate on a piece of a processor state not saved in a checkpoint have to cancel a transaction because it would be impossible to guarantee a complete state restoration otherwise. This introduces several classes of instructions that are marked as disruptive and may cancel a transaction.

- All legacy x87 FPU instructions.
- Port I/O instructions: `IN`, `OUT`.
- Instructions that change segment, debug, control, VMX, SMX or model specific registers, non-status part of EFLAGS, for example: `LDS`, far `CALL`, `VMENTER`, `CLI` etc.
- Ring transitions: `SYSENTER`, `SYSCALL`, etc. Note that it is possible to have a transaction in any ring.
- Processor state saving with `XSAVE/XRSTR`.
- Additionally, all interrupts and exceptions are required to roll back any speculation.

In our work they all were marked as disruptive. This was done to balance the implementation effort because these instructions do not give significant computational performance and in the same time increase the checkpoint size significantly.

4. Implementation

This section outlines the concepts of Simics simulation that were important for this work.

4.1. Memory transaction

Simics memory transaction is a class of objects that models CPU memory requests. A transaction contains fields that describe its properties: its logical, linear and/or physical addresses, request size, caching type (write back, write through, write combining etc), atomicity, endianness, request type (read, write, fetch, inquiry of prefetch etc.) and others. A transaction can traverse through multiple devices and its fields may be filled or modified by some of them.

4.2. Memory spaces

Real computer platforms often have a part of their physical memory space allocated for interaction with peripheral devices – so called memory mapped input/output (MMIO). To provide flexible means to model this Simics uses a concept of memory spaces – pseudodevices that map every physical address to a particular responder device. Every memory transaction originating from a processor walks through a hierarchy of memory spaces attached to it to end up in a single device which then handles it and returns the result to the requester. See Fig. 1.

Internally, a memory space stores a set of records that have these fields: **(base, length, device, priority)**. During a lookup procedure an incoming transaction address gets compared with segments of all records to find one that contains it. If there are several matches then a device with the highest priority is selected. It can turn out to be a memory space itself, in this case the lookup is repeated until a final destination is determined. A memory space can have a default target device that is used when no other mapping entry satisfies given memory transaction. A memory map contents can be dynamically changed during the simulation effectively moving, adding or hiding devices from processors.

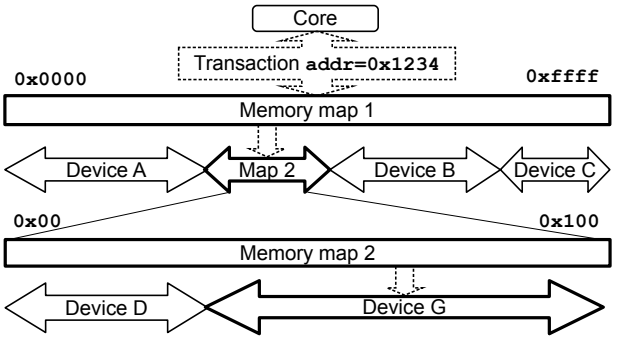

Figure 1. An example of the path of a transaction from the core through the memory space hierarchy to the device.

4.3. Translator device

The most often used Simics device classes are **ram** and **rom**, they represent continuous arrays of read-write or read-only data. They are used to model numerous RAM and EEPROM storages present in modern systems.

The most flexible class of Simics devices is called **translator**. It can implement arbitrary logic over incoming memory transactions, for example, to change its address or to multiplex responder devices. A translator device called `snare` was created to implement a transactional cache that holds speculative memory state until it is committed. Every simulated processor thread had its own instance of this device attached to it, and all snares were connected with each other in order to provide conflict detection. The main idea was to modify memory space mappings dynamically in response to events of processor threads entering or leaving speculative regions.

Let the `core[x][y]` be the *x*-th core, *y*-th thread[1] of a simulated processor that issues a memory access operation. There are three possible situations that can happen within an RTM system.

1. None of the threads are inside transactions (Fig. 2). In this the case snare is completely disabled and memory transactions are processed as usual. This guarantees that the speed of simulation of legacy workloads is unaffected.
2. `core[x][y]` is inside a speculative region, others may or may not be in their respective speculation regions (Fig. 3). The snare is active and it is inserted between the current thread's private memory space `mem[x][y]` and the shared space `phys_mem`. It redirects any incoming transactions into a new **ram** object `tx_store[x][y]` used as a cache, populating it with data in the cases of cache misses. The snare's second function is to snoop at other snares to eagerly detect read-write and write-write conflicts and notify `core[x][y]`.
3. `core[x][y]` is not using TM, but at least one remote thread is (Fig. 4). It still has to send snooping messages to the remote thread to detect memory conflicts, that is why the `snare[x][y]` is attached, though it passes incoming transactions through and does not redirect them to cache.

4.4. Implementation effort

The described solution decouples HTM logic from a processor core model which underwent minimal modifications that included decoding new instructions, saving/restoring checkpoint state and (the most time consuming and tedious) marking several classes of instructions as RTM-disruptive. The rest of the RTM logic was done in the snare device in about 2000 lines of code. We reused sources of a standard Simics cache model **g-cache**; originally it provided a reconfigurable cache with MESI coherence protocol. For our needs it was heavily modified to remove simulation of delays and to add new interfaces to interact with the processor core model; its coherence protocol was simplified in order to just detect conflicts.

While our snare model was made cache-based, it is possible to support other existing HTM protocols, for example log-based [8] or signature based [9] – only a new type of snare and no processor code modifications will be required.

[1] For models with Intel® Hyper-Threading enabled value of *y* can be either 0 or 1.

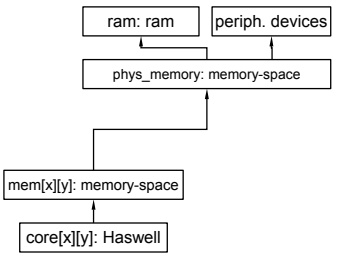

Figure 2. No threads have entered speculative execution regions. All memory transactions travel through default memory spaces to end up either in RAM or a peripheral device.

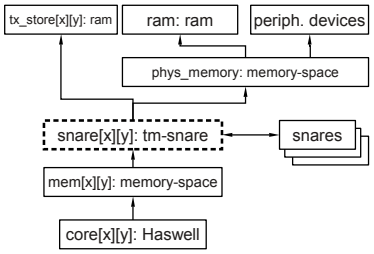

Figure 3. The thread is inside a speculative region. A new **snare** device is inserted between memory spaces to redirect relevant transactions to cache and to maintain coherency with other caches.

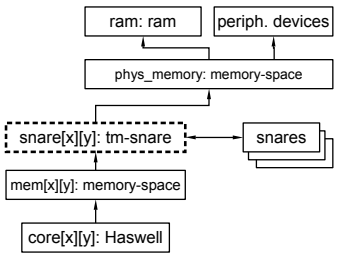

Figure 4. Current thread is not using transactional memory but a remote one is in its speculative region. The snare monitors all transactions, passes them through and notifies the remote snares.

5. Lessons learned

In this section we describe problems discovered during the model development process and in the course of its debugging in order to be able to successfully run target system and user level code.

5.1. Checking for memory caching types

It is clear that only accesses to simple, non-device backed memory can be cached inside a transaction, as delaying a MMIO request would imply its side-effects also to be delayed; that is not what a device or an OS expects. RTM specification clearly states that in such cases a transaction should be canceled and more traditional locking has to be used. In our early prototypes we did not pay great attention to this fact. Yet, it turned out that Linux kernel starts locking devices very early. Therefore we then made sure that simulated IA-32 memory type range registers (MTRRs) [10, section 11.11] purposed to track caching types of memory regions were checked to detect when a memory transaction targets a write combining or uncacheable one. Transactions that tried to access devices were constantly canceled and after several attempts they were executed non-transactionally.

5.2. Simics specifics

In the process of implementation several issues were discovered that were related to the fact that Simics sometimes uses memory transaction mechanisms to support non-architectural features. For example, non-architectural inquiries originate from the Simics integrated debugger and are meant to "peek" into the model state but not to create any architecturally visible effects.

Additionally, Simics does not expect any intermediate translation layer when setting debugger breakpoints. To make it work we did not include instruction fetches into speculative read sets and had to detect when a fetch intersects with current write set. As a consequence any events of self-modifying code inside a speculation had to cancel it. This behavior is allowed by the RTM specification: a speculative code may not be able to modify itself. For the benchmarks that we used in this work no cases of self-modifying code were observed.

5.3. VMP and transactions

Although VMP is the fastest mode available in Simics it is also known to be the most tricky one. As it relays simulation to real hardware, which can behave in subtly different ways, the results of VMP-powered simulation may depend on the used hardware. For example, virtualization features of Intel® Pentium® IV have errata that do not match Intel® Core™ family. In our case, it turned out that VMP handled most memory accesses internally without even notifying the model that they had happened; thus they could not be processed by a snare. To mitigate this we had to dynamically suppress VMP when entering speculation mode and to allow it when the last thread exits or cancels its transaction.

6. Evaluation

In this part of work a desktop with Intel i7-2600 CPU 3.40 GHz and 8 GB of RAM was used. It had 64-bit SUSE Linux Enterprise Server 10 installed as a host OS.

6.1. Applications

In order to be useful in practice new instructions should be used by applications, system libraries or operating systems. To check implementation quality and speed we evaluated two principally different simulation scenarios described below.

1. Booting of Linux kernel version 3.5.0 for 64 bit. It was modified to use lock elision unconditionally for all locks as described in [11]. Lock elision attempts to execute multiple threads in a critical sections simultaneously, each of them being in a speculative mode. If such operation turns out to be impossible due to conflicts or architectural limitations then classic locking will be used to pass that critical section.
2. STAMP [12] benchmarks version 0.9.10. This suite was designed for exploration of both software and hardware TM implementations. The tests were run under the 64-bit Ubuntu 10.04. In order to compile the benchmarks GCC version 4.5 was installed to support `asm goto` inline assembly syntax. The latter was required to define "intrinsics" showed in Fig. 5 to branch the execution to a recovery path after a transaction has been canceled. It should be noted that true RTM intrinsics have recently been added to both GNU and Intel compilers; they could have been used instead.

```
#define XBEGIN(label)   \
    asm volatile goto(".byte 0xc7,0xf8 ; \
    .long %l0-1f\n1:" ::: "eax","memory" : label)
#define XEND() asm volatile(".byte 0x0f,0x01,0xd5" ::: "memory")
#define XFAIL(label) label: asm volatile("" ::: "eax", "memory")
```

Figure 5. Macro definitions for the RTM "intrinsics"

The RTM ISA was added to the Wind River® Simics 4.6 (build 4082) model of Intel Haswell. Its core frequency was set to be 2000 MHz and number of simulated cores varied from 2 to 32 in the kernel boot experiment and from 1 to 16 for the STAMP. Prarameters of transactional caches in snares were chosen to match public specifications of level 1 data cache of Haswell, that is 32 KB of 8 ways associativity, each line 32 byte size.

As there is no guarantee that any given RTM transaction will ever progress (the real example: a Page Fault cannot be satisfied inside a transaction; only non-transactional execution of the same code is able to succeed and to clear the path for later speculative iterations) we needed an alternative non-transactional path for each region. For this we employed a single global Pthread mutex per program; it was used by a thread after several unsuccessfull attempts to elide a lock. The definitions for the common STAMP code (file `tm.h`) are shown on Fig. 6.

6.2. Simulation speed and applications behavior

For the kernel boot benchmark we studied its simulation speed measured in MIPS (that is, millions of *target* instructions per one *host* second) as reported by the Simics internal profiler `system-perfmeter`. The average MIPS were measured for an interval from the moment after the GRUB bootloader hands control over to the kernel up to the moment of mounting a root file system.

To estimate the slowdown caused by introduction of RTM three scenarios were compared: 1) RTM support was enabled and visible to the target OS and applications, all locks were attempted to be elided and VMP was active outside speculative regions; 2) RTM support was disabled, no locks were elided, while VMP was enabled all the time; 3) RTM was enabled while VMP was deactivated. The last scenario allowed to see how gravely RTM hindered benefits of VMP. As Fig. 7 shows, a simulation with RTM instructions was about three times slower than one without them. Still, until there were 32 or more simulated threads the scenario with both RTM and VMP was still faster than one with VMP off.

```
#define TM_BEGIN()  { __label__ failure;  \
                    int tries = 4;        \
                    XFAIL(failure);       \
                    tries --;             \
                    if (tries <= 0)       \
                        pthread_mutex_lock(&global_rtm_mutex);  \
                    else XBEGIN(failure);
#define TM_END()    if (tries > 0)        \
                        XEND();           \
                    else                  \
                        pthread_mutex_unlock(&global_rtm_mutex); \
                    };
```

Figure 6. Two common macro definitions of the STAMP suite rewritten to use RTM. A non standard GCC-specific `__label__` extension was used to define local block labels.

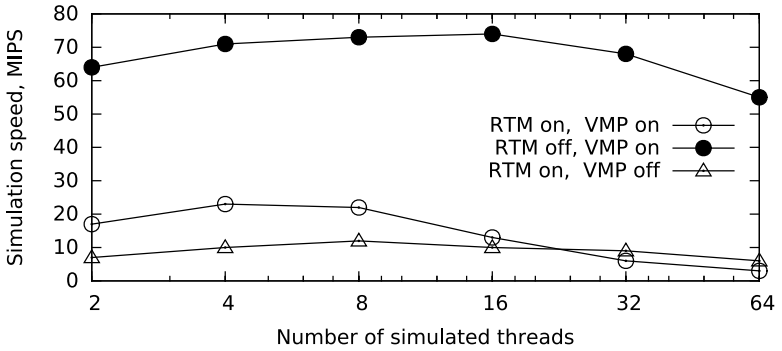

Figure 7. Simulation speeds of Linux kernel boot.

In order for this Simics model to be useful for software developers it should be possible to monitor how well RTM code performs in applications, at least in terms of frequency of conflicts and other events disruptive to the speculative execution. To study this we analyzed STAMP benchmarks.

Input parameters for individual tests were taken from README files accompanying each of them as suggested for simulator runs (Tab. 1). Although we were able to compile all programs, only seven out of eight managed to run successfully – the **labyrinth** program reported a failed assertion error during our tests and was not included in the results.

A possibility to monitor frequency of RTM events was added to the simulator by means of a set of counters that can be accessed through the Simics command line interface. Fig. 8 shows collected statistics for the kernel and STAMP workloads. The following types of events that conclude individual transactions were recorded: *committed* – successfully finished, *capacity* – canceled due to cache overflow, *conflict* – two or more threads accessed the same line, *exception* – an exception occured while in speculation, *instruction* – a disruptive instruction that cannot be executed inside a transaction encountered, *interrupt* – an external interrupt forced a transaction to be canceled.

The following observations can be made out of this experiment.

- The kernel boot test was the one with the highest ratio of transactions canceled due to disruptive instructions `IN/OUT` and other privileged ones. This highlights the fact that not all locks can or should be attempted to be elided, especially in privileged software that is able to execute more classes of disruptive instructions.
- Characterizations of individual STAMP benchmarks given in [12] matched behavior observed in our tests. For example, the least contended ones were **kmeans** and **ssca2**.

Fig. 9 shows simulation speeds for STAMP benchmarks. As expected, simulation of user level applications was

Table 1. STAMP benchmarks command line arguments used in simulation.

Benchmark name	Command line arguments
bayes	-v32 -r1024 -n2 -p20 -s0 -i2 -e2 -t 16
genome	-g256 -s16 -n16384 -t 16
intruder	-a10 -l4 -n2038 -s1 -t 16
kmeans	-m15 -n15 -t0.05 -i inputs/random-n2048-d16-c16.txt -p 16
labyrinth	*N/A – failed to run*
ssca2	-s13 -i1.0 -u1.0 -l3 -p3 -t 16
vacation	-n2 -q90 -u98 -r16384 -t4096 -c 16
yada	-a20 -i inputs/633.2 -t 16

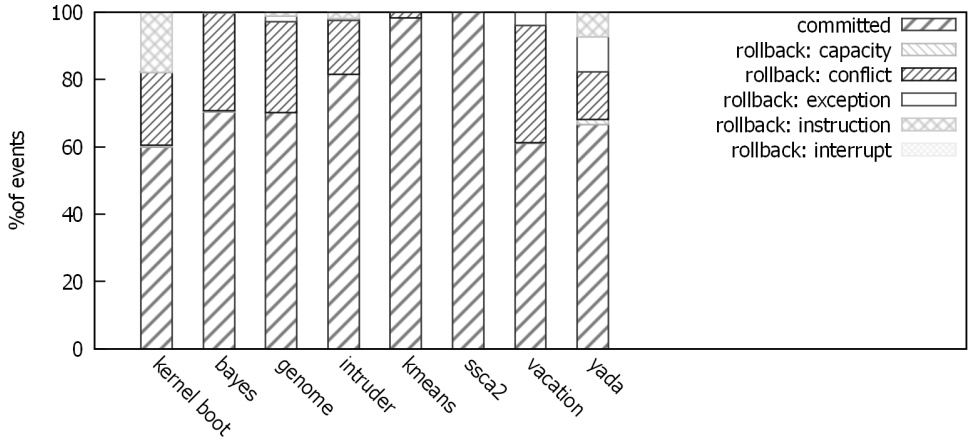

Figure 8. Outcomes of transactions for studied benchmarks.

overall faster than system level kernel code because less complex privileged instructions were encountered and more code could be run in VMP.

7. Related work

7.1. Hardware support

As it has been said in the introduction to this paper, IBM was the first to implement TM in hardware in the BlueGene/Q system processor. AMD has announced an Advanced Synchronization Facility (ASF) [13] ISA extension proposal that constitutes another variant of HTM for the IA-32 architecture. Public documentation exists but no hardware has been announced yet. Sun developed a processor codenamed ROCK with HTM support. After the company was acquired by Oracle the ROCK project was canceled.

7.2. Software and simulator support

Too many variants of software TM implementations exist to mention them all. Generally they do not require special hardware support from a CPU. Unfortunately, current observations are that the overhead of purely software implementations severely diminishes benefits of TM programmability. Therefore we focused on comparison of our work with software simulation solutions which were created for different architectures and were based on different software products.

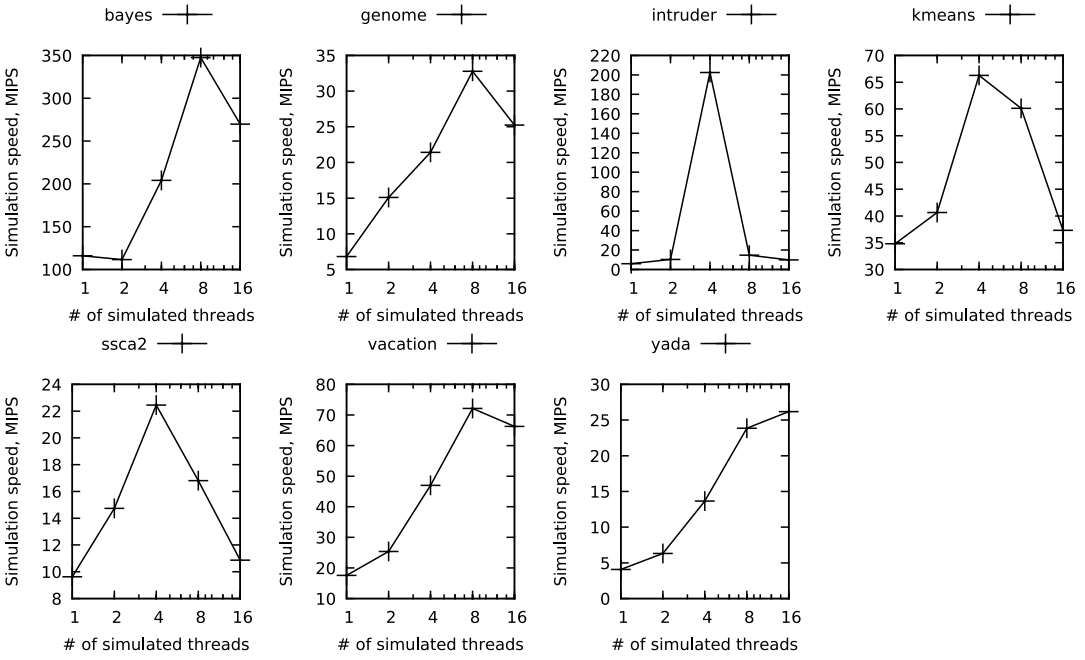

Figure 9. Simulation speeds of STAMP benchmarks.

A number of HTM studies exist that leverage Simics with GEMS [14] – a specialized cache simulator. Liu et al. [15] propose an unofficial extension for the SPARC architecture and are mainly interested in nested transactions mechanics. Yen et al. [8], who extend SPARC as well, devised a spectrum of sophisticated HTM systems and give a thorough analysis of their performance. Sun ROCK simulator support for HTM is described in Moir et al. [16], their work was also based on Simics/GEMS and was for SPARC.

Compared to these papers in our work we focused on the officially announced technology and our target architecture was IA-32. Similarly, AMD ASF is for AMD64 extension to IA-32; it was implemented and studied in the out-of-order cycle precise simulator PTLSim [17].

Considering Intel TSX alone, there is another simulator Intel® SDE [18] that supports the new ISA. Being application mode only, it is unable to evaluate operation of RTM in system software such as Linux.

8. Conclusions and future work

In this paper we described modifications to the Wind River® Simics required to add support of Intel® Restricted Transactional Memory instruction set architecture extension. The initial goal to have a minimal impact on the simulation speed was accomplished by carrying the transactional cache out to a separate device from the processor model. Then we verified that our implementation is correct enough to execute both operating system and user level workloads. It was also shown that the simulation speed was high enough to be useful for software developement using this model. The results of this work were included in the Wind River® Simics Haswell model package. The modifications to STAMP for RTM support were published at Github (`https://github.com/grigory-rechistov/stamp-rtm`).

It should be noted that currently RTM instructions are only interpreted by Simics, no JIT mode has been enabled for them. This negatively affects simulation performance. Our plans include enabling of JIT mode – it should not require drastic changes to Simics core as it shares a lot of common code with interpreter. We also expect to carry out a more extensive verification of the implementation correctness on a broader set of user applications, libraries and operating systems.

Finally, it should be noted that the second part of Intel TSX documentation describes a Hardware Lock Elision (HLE) extension that introduces another, backwards binary compatible yet somewhat less explicit method to elide

locks with help of two IA-32 instruction prefixes. It is planned to leverage the approach described in this work to model HLE as well.

Acknowledgements

Authors would like to thank Alexander Ivanov, Oleg Oleinik, Alexey Kovalev, Daniel Aarno and anonymous reviewers for their valuable comments on the paper.

References

[1] M. Herlihy, J. E. B. Moss, Transactional memory: architectural support for lock-free data structures, in: Proceedings of the 20th annual international symposium on computer architecture, ISCA '93, ACM, New York, NY, USA, 1993, pp. 289–300. doi:10.1145/165123.165164.
URL http://doi.acm.org/10.1145/165123.165164

[2] R. Merritt, IBM plants transactional memory in CPU, EE Times.
URL http://www.eetimes.com/electronics-news/4218914/IBM-plants-transactional-memory-in-CPU

[3] IBM unveils zEnterprise EC12, a highly secure system for cloud computing and enterprise data (August 2012).
URL http://www-03.ibm.com/press/us/en/pressrelease/38653.wss

[4] R. Rajwar, M. Dixon, Intel transactional synchronization extensions (2012).
URL http://intel.com/go/idfsessions

[5] Intel Corporation, Intel Architecture Instruction Set Extensions Programming Reference (July 2012).
URL http://software.intel.com/file/41417

[6] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Hgberg, F. Larsson, A. Moestedt, B. Werner, Simics: A Full System Simulation Platform, Computer 35 (2002) 5058. doi:10.1109/2.982916.
URL http://portal.acm.org/citation.cfm?id=619072.621909

[7] F. Leung, G. Neiger, D. Rodgers, A. Santoni, R. Uhlig, Intel Virtualization Technology, Intel Technology Journal 10. doi:10.1535/itj.1003.01.
URL http://www.intel.com/technology/itj/2006/v10i3/

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, Logtm: Log-based transactional memory, in: HPCA, 2006, pp. 254–265.

[9] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, LogTM-SE: Decoupling Hardware Transactional Memory from Caches, The 13th Annual International Symposium on High Performance Computer Architecture (HPCA-13).
URL http://research.cs.wisc.edu/multifacet/papers/hpca07_logtmse.pdf

[10] Intel Corporation, Intel 64 and IA-32 Architectures Software Developers Manual. Volume 3A (2012).
URL http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[11] A. Kleen, Adding lock elision to Linux (August 2012).
URL http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/Adding-lock-elision-to-Linux.pdf

[12] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization.

[13] J. Chung, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, D. Grossman, ASF: AMD64 extension for lock-free data structures and transactional memory (2010).

[14] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, SIGARCH Comput. Archit. News 33 (4) (2005) 92–99. doi:10.1145/1105734.1105747.
URL http://doi.acm.org/10.1145/1105734.1105747

[15] Y. Liu, Y. Su, C. Zhang, M. Wu, X. Zhang, H. Li, D. Qian, Efficient transaction nesting in hardware transactional memory, in: Proceedings of the 23rd international conference on Architecture of Computing Systems, ARCS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 138–149. doi:10.1007/978-3-642-11950-7_13.
URL http://dx.doi.org/10.1007/978-3-642-11950-7_13

[16] M. Moir, K. Moore, D. Nussbaum, The adaptive transactional memory test platform: a tool for experimenting with transactional code for ROCK (poster), in: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, SPAA '08, ACM, New York, NY, USA, 2008, pp. 362–362. doi:10.1145/1378533.1378595.
URL http://doi.acm.org/10.1145/1378533.1378595

[17] S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, J.-W. Chung, L. Yen, Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core, 5th ACMSIGPLAN Workshop on Transactional Computing.
URL http://www.amd64.org/fileadmin/user_upload/pub/transact10-asfooo.pdf

[18] Intel Software Development Emulator.
URL http://software.intel.com/en-us/articles/intel-software-development-emulator